# Golang日志库 log

## 什么是日志

所谓日志（Log）是指系统所指定对象的某些操作和其操作结果按时间有序的集合。log文件就是日志文件，log文件记录了系统和系统的用户之间交互的信息，是自动捕获人与系统终端之间交互的类型、内容或时间的数据收集方法；

日志是用来记录，用户操作，系统状态，错误信息等等内容的文件，是一个软件系统的重要组成部分。一个良好的日志规范，对于系统运行状态的分析，以及线上问题的解决具有重大的意义。

**日志规范**

在开发软件打印日志时，需要注意一些问题，举例可能不全

* 重要功能日志尽可能的完善。
* 不要随意打印无用的日志，过多无用的日志会增加分析日志的难度。
* 日志要区分等级 如 debug，warn，info，error 等。
* 捕获到未处理错误时最好打印错误堆栈信息

### Go 语言常用的日志库

Go 语言标准库中就为我们提供了一个日志库 log，除了这个以外还有很多日志库，如 logrus，glog，logx，Uber 的 zap 等等。

在日常开发中，日志是必不可少的功能。虽然有时可以用fmt库输出一些信息，但是灵活性不够。Go 标准库提供了一个日志库log。本文介绍log库的使用。

## 快速使用

log是 Go 标准库提供的，不需要另外安装。可直接使用：

package main  
  
import (  
 "log"  
)  
  
type User struct {  
 Name string  
 Age int  
}  
  
func main() {  
 u := User{  
 Name: "dj",  
 Age: 18,  
 }  
  
 log.Printf("%s login, age:%d", u.Name, u.Age)  
 log.Panicf("Oh, system error when %s login", u.Name)  
 log.Fatalf("Danger! hacker %s login", u.Name)  
}

log默认输出到标准错误（stderr），每条日志前会自动加上日期和时间。如果日志不是以换行符结尾的，那么log会自动加上换行符。即每条日志会在新行中输出。

log提供了三组函数：

* Print/Printf/Println：正常输出日志；
* Panic/Panicf/Panicln：输出日志后，以拼装好的字符串为参数调用panic；
* Fatal/Fatalf/Fatalln：输出日志后，调用os.Exit(1)退出程序。

命名比较容易辨别，带f后缀的有格式化功能，带ln后缀的会在日志后增加一个换行符。

注意，上面的程序中由于调用log.Panicf会panic，所以log.Fatalf并不会调用。

## 定制

### 前缀

调用log.SetPrefix为每条日志文本前增加一个前缀。例如，在上面的程序中设置Login:前缀：

package main  
  
import (  
 "log"  
)  
  
type User struct {  
 Name string  
 Age int  
}  
  
func main() {  
 u := User{  
 Name: "dj",  
 Age: 18,  
 }  
  
 log.SetPrefix("Login: ")  
 log.Printf("%s login, age:%d", u.Name, u.Age) //输出：Login: 2022/12/18 15:35:56 dj login, age:18  
}

调用log.Prefix可以获取当前设置的前缀。

### 选项

设置选项可在每条输出的文本前增加一些额外信息，如日期时间、文件名等。

log库提供了 6 个选项：

// src/log/log.go  
const (  
 *Ldate* = 1 << iota  
 *Ltime  
 Lmicroseconds  
 Llongfile  
 Lshortfile  
 LUTC*)

* Ldate：输出当地时区的日期，如2020/02/07；
* Ltime：输出当地时区的时间，如11:45:45；
* Lmicroseconds：输出的时间精确到微秒，设置了该选项就不用设置Ltime了。如11:45:45.123123；
* Llongfile：输出长文件名+行号，含包名，如github.com/darjun/go-daily-lib/log/flag/main.go:50；
* Lshortfile：输出短文件名+行号，不含包名，如main.go:50；
* LUTC：如果设置了Ldate或Ltime，将输出 UTC 时间，而非当地时区。

调用log.SetFlag设置选项，可以一次设置多个：

package main  
  
import (  
 "log"  
)  
  
type User struct {  
 Name string  
 Age int  
}  
  
func main() {  
 u := User{  
 Name: "dj",  
 Age: 18,  
 }  
  
 log.SetFlags(log.*Lshortfile* | log.*Ldate* | log.*Lmicroseconds*)  
  
 log.Printf("%s login, age:%d", u.Name, u.Age) //输出：2022/12/18 15:59:12.587327 example.go:20: dj login, age:18  
}

调用log.Flags()可以获取当前设置的选项。

运行代码，输出：

2020/02/07 11:56:59.061615 main.go:20: dj login, age:18

注意，调用log.SetFlag之后，原有的选项会被覆盖掉！

log库还定义了一个Lstdflag，为Ldate | Ltime，这就是我们默认的选项。

// src/log/log.go  
const (  
 *LstdFlags* = Ldate | Ltime  
)

这就是为什么默认情况下，每条日志前会自动加上日期和时间。

### 自定义

实际上，log库为我们定义了一个默认的Logger，名为std，意为标准日志。我们直接调用的log库的方法，其内部是调用std的对应方法：

var std = New(os.Stderr, "", *LstdFlags*)

// Printf calls Output to print to the standard logger.  
// Arguments are handled in the manner of fmt.Printf.  
func Printf(format string, v ...any) {  
 if atomic.LoadInt32(&std.isDiscard) != 0 {  
 return  
 }  
 std.Output(2, fmt.Sprintf(format, v...))  
}

// Fatalf is equivalent to Printf() followed by a call to os.Exit(1).  
func Fatalf(format string, v ...any) {  
 std.Output(2, fmt.Sprintf(format, v...))  
 os.Exit(1)  
}

当然，我们也可以定义自己的Logger：

package main  
  
import (  
 "bytes"  
 "fmt"  
 "log"  
)  
  
type User struct {  
 Name string  
 Age int  
}  
  
func main() {  
 u := User{  
 Name: "dj",  
 Age: 18,  
 }  
  
 buf := &bytes.Buffer{}  
 logger := log.New(buf, "", log.*Lshortfile*|log.*LstdFlags*)  
  
 logger.Printf("%s login, age:%d", u.Name, u.Age)  
  
 fmt.Print(buf.String())  
}

log.New接受三个参数：

* io.Writer：日志都会写到这个Writer中；
* prefix：前缀，也可以后面调用logger.SetPrefix设置；
* flag：选项，也可以后面调用logger.SetFlag设置。

上面代码将日志输出到一个bytes.Buffer，然后将这个buf打印到标准输出。

运行代码：

注意到，第一个参数为io.Writer，我们可以使用io.MultiWriter实现多目的地输出。下面我们将日志同时输出到标准输出、bytes.Buffer和文件中：

$ go run main.go   
2020/02/07 13:48:54 main.go:23: dj login, age:18

如果你愿意，还可以发送到到网络。

## **实现**

log库的核心是Output方法，我们简单看一下：

// Output writes the output for a logging event. The string s contains  
// the text to print after the prefix specified by the flags of the  
// Logger. A newline is appended if the last character of s is not  
// already a newline. Calldepth is used to recover the PC and is  
// provided for generality, although at the moment on all pre-defined  
// paths it will be 2.  
func (l \*Logger) Output(calldepth int, s string) error {  
 now := time.Now() // get this early.  
 var file string  
 var line int  
 l.mu.Lock()  
 defer l.mu.Unlock()  
 if l.flag&(*Lshortfile*|*Llongfile*) != 0 {  
 // Release lock while getting caller info - it's expensive.  
 l.mu.Unlock()  
 var ok bool  
 \_, file, line, ok = runtime.Caller(calldepth)  
 if !ok {  
 file = "???"  
 line = 0  
 }  
 l.mu.Lock()  
 }  
 l.buf = l.buf[:0]  
 l.formatHeader(&l.buf, now, file, line)  
 l.buf = append(l.buf, s...)  
 if len(s) == 0 || s[len(s)-1] != '\n' {  
 l.buf = append(l.buf, '\n')  
 }  
 \_, err := l.out.Write(l.buf)  
 return err  
}

如果设置了Lshortfile或Llongfile，Ouput方法中会调用runtime.Caller获取文件名和行号。runtime.Caller的参数calldepth表示获取调用栈向上多少层的信息，当前层为 0。

一般的调用路径是：

* 程序中使用log.Printf之类的函数；
* 在log.Printf内调用std.Output。

我们在Output方法中需要获取调用log.Printf的文件和行号。 calldepth传入 0 表示Output方法内调用runtime.Caller的那一行信息，传入 1 表示log.Printf内调用std.Output那一行的信息， 传入 2 表示程序中调用log.Printf的那一行信息。显然这里要用 2。

然后调用formatHeader处理前缀和选项。

最后将生成的字节流写入到Writer中。

这里有两个优化技巧：

* 由于runtime.Caller调用比较耗时，先释放锁，避免等待时间过长；
* 为了避免频繁的内存分配，logger中保存了一个类型为[]byte的buf，可重复使用。前缀和日志内容先写到这个buf中，然后统一写入Writer，减少 io 操作。